Order Management System design

Order Management System - High Level Design Document

Version: 1.0

Author: Somanath Kambar

Executive Summary

This document outlines the High-Level Design (HLD) for the Order Management System (OMS), a scalable, multi-tenant e-commerce platform built using Spring Boot. The system is designed to evolve through three distinct phases, supporting growth from startup to enterprise scale while maintaining reliability, performance, and security.

Table of Contents

1. INTRODUCTION & VISION

1.1 Business Context

The Order Management System is a core e-commerce platform that enables businesses to manage their entire order lifecycle from creation to fulfillment. The system supports multi-tenancy for white-label deployments across different business verticals.

1.2 Goals & Objectives

- Primary Goal: Build a scalable order processing system handling 1M+ daily orders
- Reliability: 99.95% availability with robust failure handling
- Performance: <200ms response time for 95th percentile of requests
- Scalability: Support growth from 100 to 1,000,000 users
- Security: PCI-DSS compliance for payment processing
- Multi-tenancy: White-label support for different business brands

Requirement	Target	Measurement
Availability	99.95%	Monthly uptime percentage
Latency	<200ms	P95 response time
Throughput	1000 orders/sec	Peak capacity
Data Retention	7 years	Regulatory compliance
Recovery Time	<1 hour	RTO (Recovery Time Objective)
Recovery Point	<5 minutes	RPO (Recovery Point Objective)

Included:

- Order lifecycle management (create, update, cancel, fulfill)
- Inventory integration and reservation
- Payment processing workflows
- Customer management
- Multi-tenant isolation
- Analytics and reporting

Excluded:

- Product catalog management (separate service)
- User authentication/authorization (external IAM)
- Warehouse management systems (WMS integration only)
- Shipping carrier integration (external APIs)
- Marketing and promotions (separate service)

2. ARCHITECTURE OVERVIEW

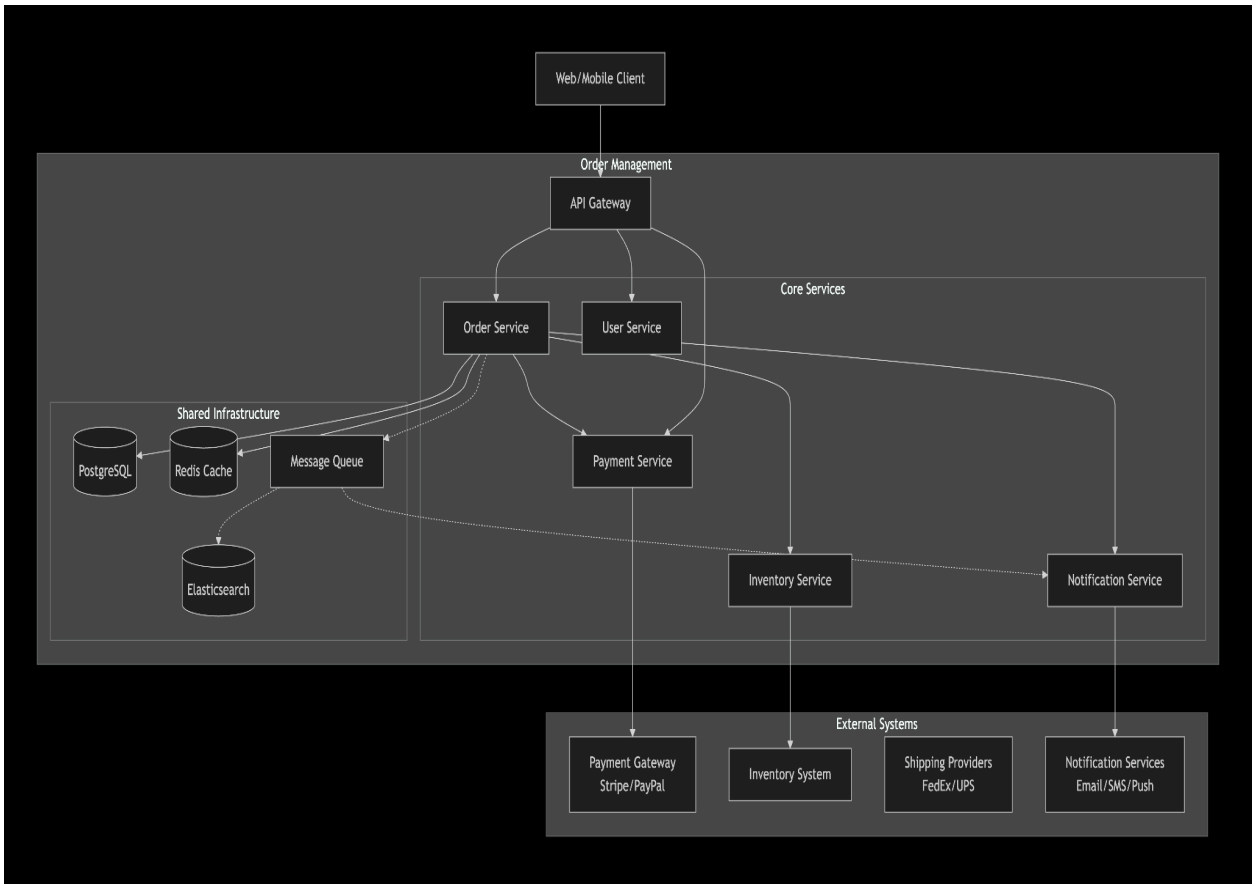
2.1 Architectural Principles

1. Evolutionary Architecture: Start simple, scale gradually
2. Domain-Driven Design: Business logic at the core
3. Clean Architecture: Separation of concerns
4. Event-Driven: Asynchronous processing where possible
5. Resilience: Circuit breakers, retries, fallbacks
6. Observability: Comprehensive monitoring from day one

2.2 Technology Stack

Layer	Phase 1	Phase 2	Phase 3
Application	Spring Boot Monolith	Spring Boot + Spring Cloud	Spring Boot Microservices
Database	PostgreSQL (single)	PostgreSQL (primary+replicas)	Polyglot (PostgreSQL, Cassandra, Redis)
Cache	Caffeine (in-memory)	Redis (single)	Redis Cluster
Message Queue	In-memory/Spring Events	RabbitMQ	Apache Kafka
API Gateway	Spring MVC	Spring Cloud Gateway	Kong/Envoy
Service Registry	-	Eureka/Consul	Consul + Istio
Monitoring	Spring Actuator	Prometheus + Grafana	ELK + Jaeger + Prometheus

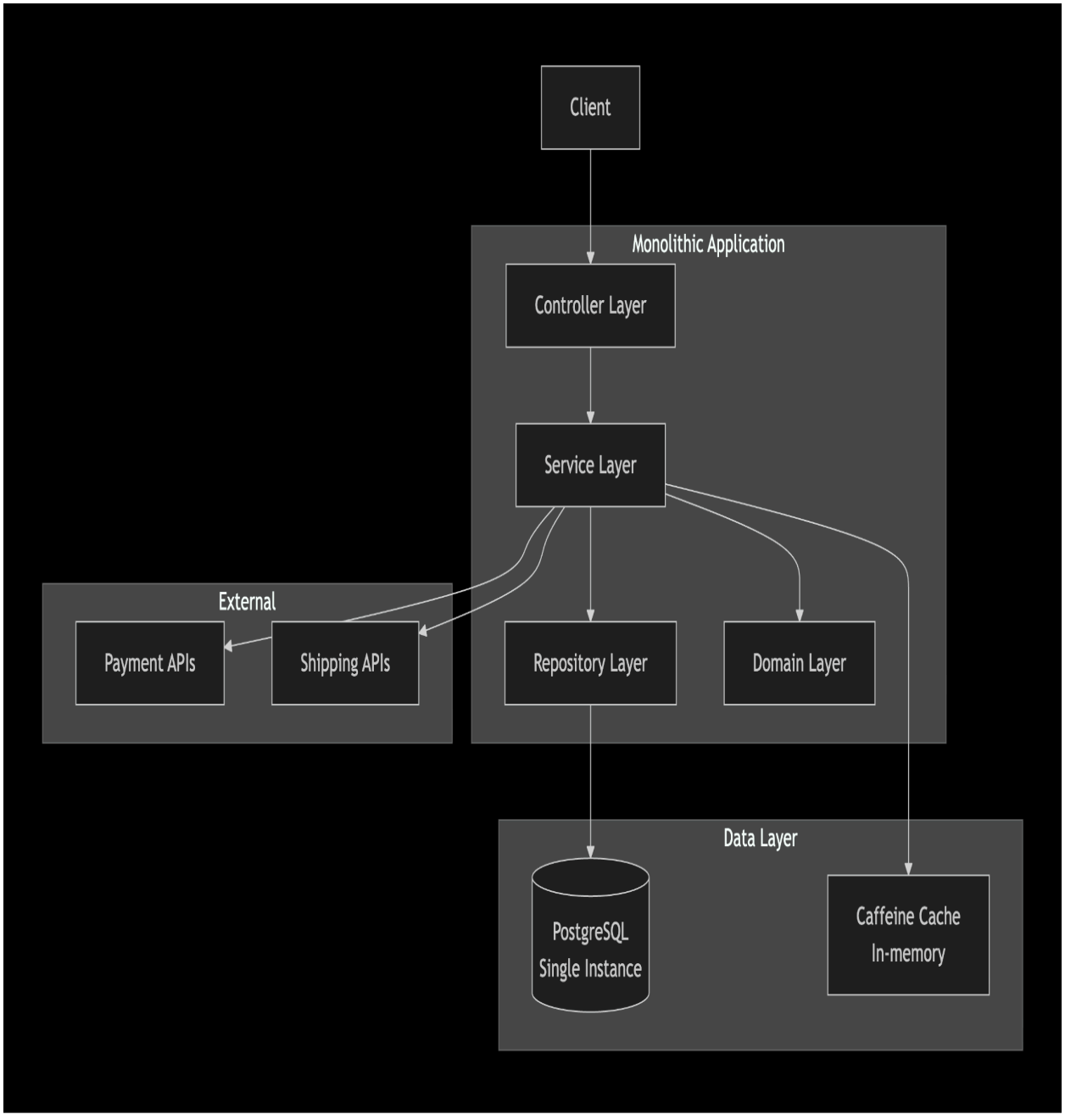
2.3 High-Level Component Diagram



Characteristics:

- Single deployable unit
- Modular packages within monolith
- In-memory caching
- Direct external API calls
- Single database instance

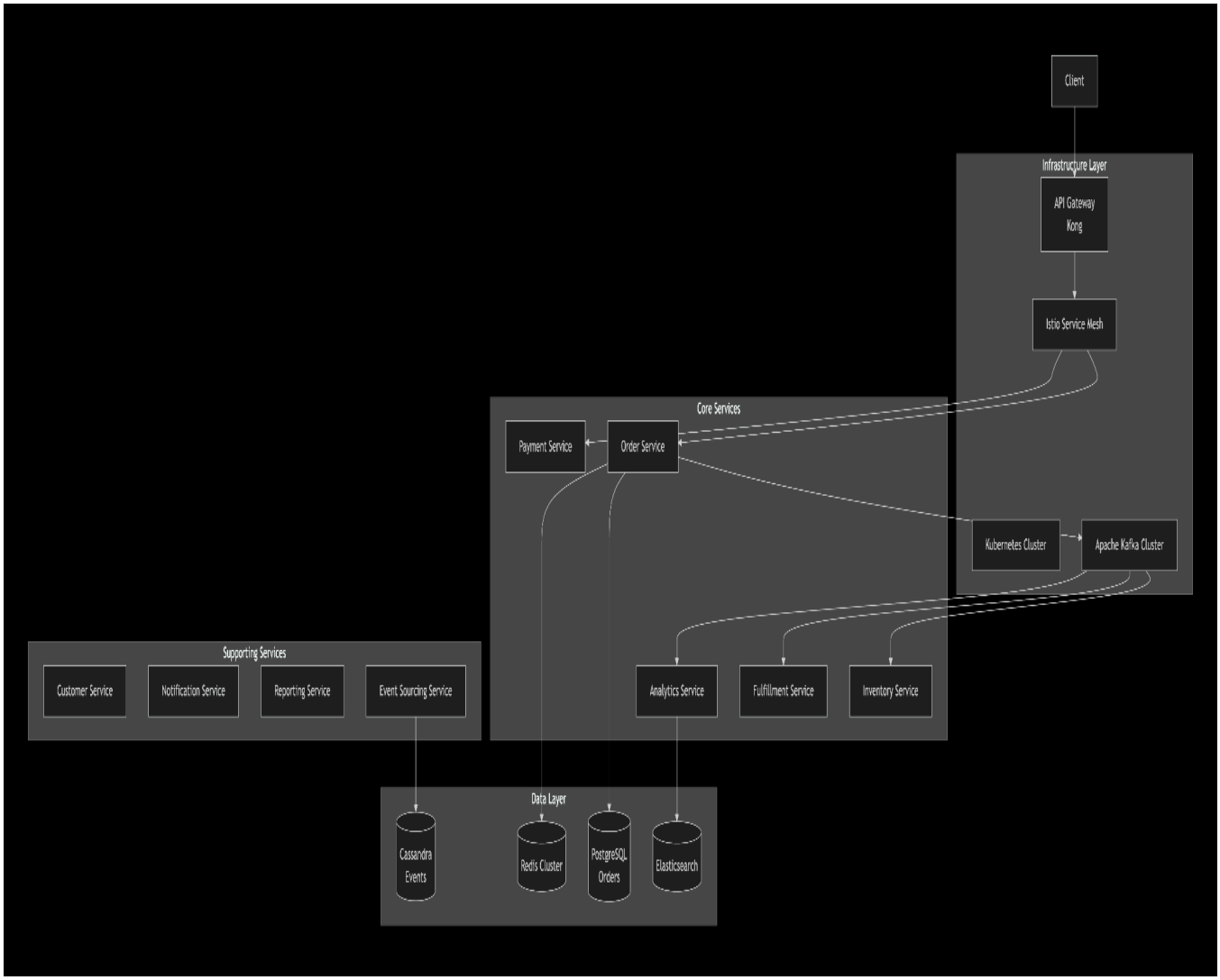
3.1 Phase 1: Monolithic Architecture (0-10K Users)



3.2 Phase 2: Service-Oriented Architecture (10K-100K Users)
Characteristics:

- API Gateway for routing
- Service discovery
- Separate databases per service
- Message queue for async processing
- Read replicas for scaling

3.3 Phase 3: Microservices + Event-Driven (100K-1M+ Users)



Characteristics:

- Kubernetes orchestration
- Service mesh for communication
- Event streaming with Kafka
- Polyglot persistence
- Global distribution

4. DATABASE DESIGN

4.1 Entity Relationship Diagram (ERD)

4.2 Table Definitions

Orders Table (Partitioned by Date) -- Partitioned table for orders

```
CREATE TABLE orders (
    id UUID DEFAULT gen_random_uuid() PRIMARY KEY,
    tenant_id UUID NOT NULL REFERENCES tenants(id),
    customer_id UUID NOT NULL REFERENCES customers(id),
    order_number VARCHAR(50) NOT NULL UNIQUE,
    status VARCHAR(20) NOT NULL DEFAULT 'DRAFT'
        CHECK (status IN ('DRAFT', 'CONFIRMED', 'PAID', 'PROCESSING',
                          'SHIPPED', 'DELIVERED', 'CANCELLED', 'REFUNDED')),
    total_amount DECIMAL(10,2) NOT NULL CHECK (total_amount >= 0),
    tax_amount DECIMAL(10,2) NOT NULL DEFAULT 0,
    shipping_amount DECIMAL(10,2) NOT NULL DEFAULT 0,
    currency VARCHAR(3) NOT NULL DEFAULT 'USD',
    shipping_address JSONB NOT NULL,
    billing_address JSONB,
    metadata JSONB, -- Flexible field for tenant-specific data
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    -- Indexes
    INDEX idx_orders_tenant_status (tenant_id, status),
    INDEX idx_orders_customer (customer_id),
    INDEX idx_orders_created (created_at DESC)
) PARTITION BY RANGE (created_at);
-- Create monthly partitions
CREATE TABLE orders_2024_01 PARTITION OF orders
    FOR VALUES FROM ('2024-01-01') TO ('2024-02-01');
CREATE TABLE orders_2024_02 PARTITION OF orders
    FOR VALUES FROM ('2024-02-01') TO ('2024-03-01');
```

Order Items Table

```
CREATE TABLE order_items (
    id UUID DEFAULT gen_random_uuid() PRIMARY KEY,
    order_id UUID NOT NULL REFERENCES orders(id) ON DELETE CASCADE,
    product_id UUID NOT NULL,
    sku VARCHAR(100) NOT NULL,
    name VARCHAR(255) NOT NULL,
    quantity INTEGER NOT NULL CHECK (quantity > 0),
    unit_price DECIMAL(10,2) NOT NULL CHECK (unit_price >= 0),
    total_price DECIMAL(10,2) GENERATED ALWAYS AS (quantity * unit_price) STORED,
    metadata JSONB,

    INDEX idx_order_items_order (order_id),
    INDEX idx_order_items_product (product_id)
);
```

Inventory Table with Optimistic Locking

```
CREATE TABLE inventory (
    id UUID DEFAULT gen_random_uuid() PRIMARY KEY,
    tenant_id UUID NOT NULL REFERENCES tenants(id),
    product_id UUID NOT NULL,
    warehouse_id UUID NOT NULL,
    available_quantity INTEGER NOT NULL DEFAULT 0 CHECK (available_quantity >= 0),
    reserved_quantity INTEGER NOT NULL DEFAULT 0 CHECK (reserved_quantity >= 0),
    sold_quantity INTEGER NOT NULL DEFAULT 0 CHECK (sold_quantity >= 0),
    version INTEGER NOT NULL DEFAULT 0, -- For optimistic locking
    last_updated TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,

    UNIQUE (tenant_id, product_id, warehouse_id),
    INDEX idx_inventory_product (product_id),
    INDEX idx_inventory_tenant (tenant_id)
);
```

```
-- Phase 1: Single Database
-- All tables in one schema with tenant_id for isolation
```

```
-- Phase 2: Schema per Tenant (for enterprise customers)
CREATE SCHEMA tenant_acme;
CREATE SCHEMA tenant_globex;
-- Phase 3: Database per Tenant (for maximum isolation)
-- Separate RDS instances for large tenants
-- Migration script example
DO $$
BEGIN
   -- Add new column without downtime
   IF NOT EXISTS (SELECT 1 FROM information_schema.columns
                  WHERE table_name = 'orders'
                  AND column_name = 'fulfillment_center_id') THEN
      ALTER TABLE orders ADD COLUMN fulfillment_center_id UUID;
   END IF;
END $$;
```

4.3 Database Evolution Strategy


```
-- Phase 1: Single Database
-- All tables in one schema with tenant_id for isolation
-- Phase 2: Schema per Tenant (for enterprise customers)
CREATE SCHEMA tenant_acme;
CREATE SCHEMA tenant_globex;
-- Phase 3: Database per Tenant (for maximum isolation)
-- Separate RDS instances for large tenants
-- Migration script example
DO $$
BEGIN
   -- Add new column without downtime
   IF NOT EXISTS (SELECT 1 FROM information_schema.columns
                  WHERE table_name = 'orders'
                  AND column_name = 'fulfillment_center_id') THEN
      ALTER TABLE orders ADD COLUMN fulfillment_center_id UUID;
   END IF;
END $$;
```

5. API DESIGN

5.1 REST API Endpoints

Resource	Method	Endpoint	Description	Auth Required
Orders	GET	/api/v1/orders	List orders with filters	Yes
	POST	/api/v1/orders	Create new order	Yes
	GET	/api/v1/orders/{id}	Get order by ID	Yes
	PUT	/api/v1/orders/{id}	Update order	Yes
	PATCH	/api/v1/orders/{id}/status	Update order status	Yes
	DELETE	/api/v1/orders/{id}	Cancel order	Yes
Payments	POST	/api/v1/orders/{id}/payments	Process payment	Yes
	GET	/api/v1/orders/{id}/payments	Get payment history	Yes
Fulfillment	POST	/api/v1/orders/{id}/shipments	Create shipment	Yes
	GET	/api/v1/orders/{id}/tracking	Get tracking info	Yes

5.2 API Versioning Strategy

yaml
API Versioning: URL Path versioning
Example:
- /api/v1/orders (Current)
- /api/v2/orders (Future, backward compatible)

Deprecation Policy:
- Version supported for 18 months after new release
- Deprecation warning in headers for 6 months
- Breaking changes only in major versions

5.3 Request/Response Examples

Create Order Request

```
POST /api/v1/orders
Content-Type: application/json
X-Tenant-Id: acme-corp
Authorization: Bearer <jwt-token>
{
 "customerId": "cust_12345",
 "items": [
  {
   "productId": "prod_67890",
   "sku": "IPHONE-13-PRO",
   "quantity": 1,
   "price": 999.99
  }
 ],
 "shippingAddress": {
  "street": "123 Main St",
  "city": "San Francisco",
  "state": "CA",
  "zipCode": "94105",
  "country": "US"
 },
 "paymentMethod": {
  "type": "credit_card",
  "token": "pm_1Jk123456789"
 }
}
```

Create Order Response


```
{
 "orderId": "ord_abc123def456",
 "orderNumber": "ORD-2024-001234",
 "status": "CONFIRMED",
 "totalAmount": 1079.98,
 "items": [
  {
   "productId": "prod_67890",
   "name": "iPhone 13 Pro",
   "quantity": 1,
   "unitPrice": 999.99,
   "totalPrice": 999.99
  }
 ],
 "estimatedDelivery": "2024-01-10",
 "paymentStatus": "PROCESSING",
 "links": {
  "self": "/api/v1/orders/ord_abc123def456",
  "payment": "/api/v1/orders/ord_abc123def456/payments",
  "tracking": "/api/v1/orders/ord_abc123def456/tracking"
 }
}
```


5.4 Error Handling

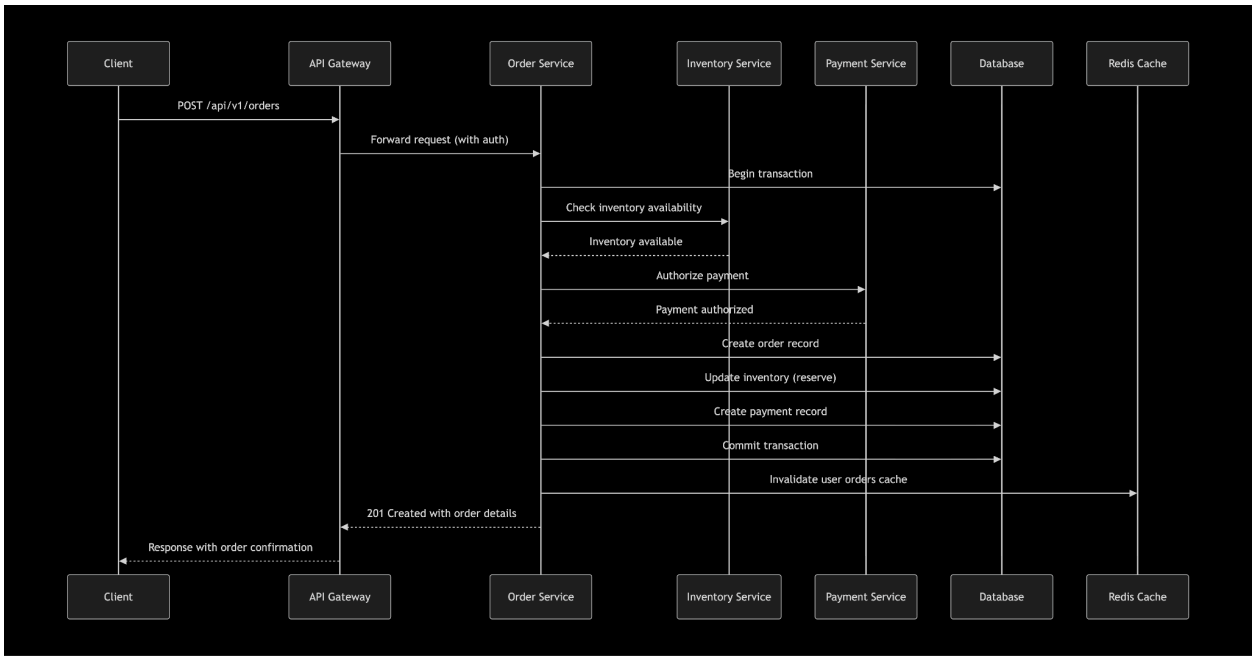

```
{
 "error": {
  "code": "ORDER_NOT_FOUND",
  "message": "Order with ID 'ord_123' not found",
  "details": "The requested order does not exist or you don't have permission to access
it",
  "timestamp": "2024-12-25T10:30:00Z",
  "requestId": "req_987654321",
  "documentation": "https://docs.example.com/errors/ORDER_NOT_FOUND"
 }
}
```


Standard HTTP Status Codes:

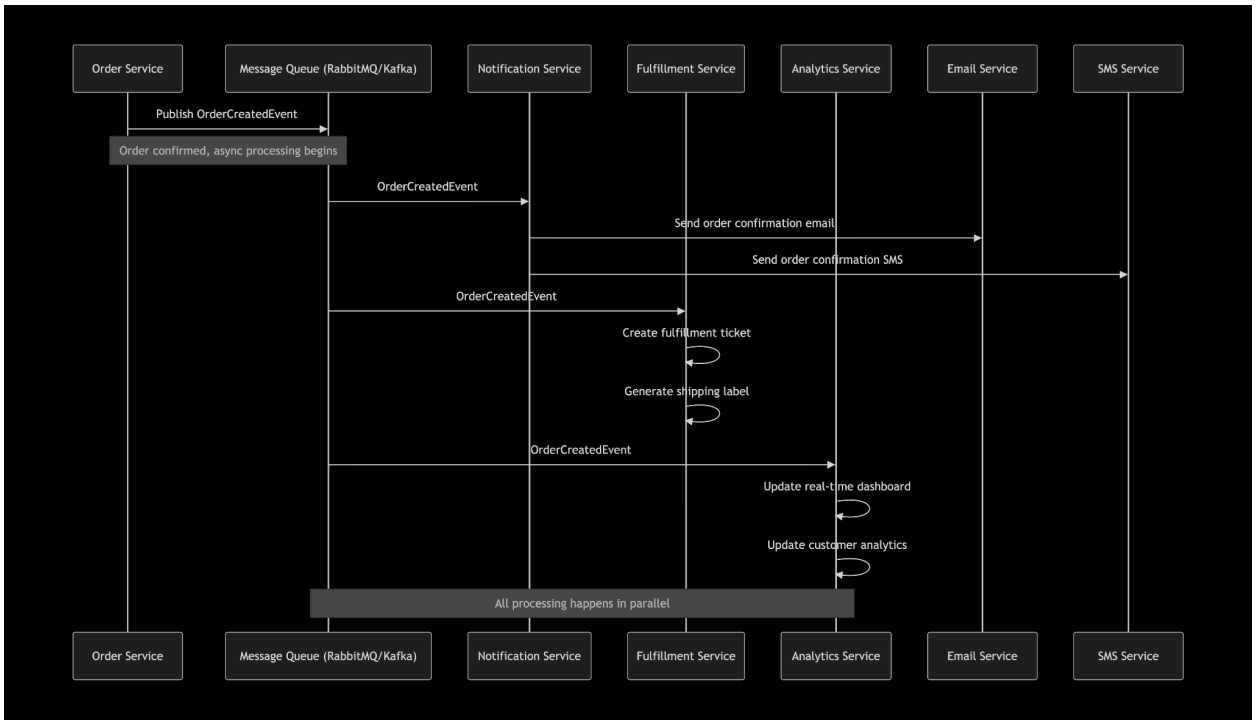
- 200 OK: Successful GET/PUT/PATCH
- 201 Created: Successful POST
- 400 Bad Request: Invalid input
- 401 Unauthorized: Authentication required
- 403 Forbidden: Insufficient permissions
- 404 Not Found: Resource not found
- 409 Conflict: State conflict (e.g., order already shipped)
- 422 Unprocessable Entity: Business validation failed
- 429 Too Many Requests: Rate limit exceeded
- 500 Internal Server Error: Server error
- 503 Service Unavailable: Service temporarily unavailable

6. COMPONENT INTERACTIONS

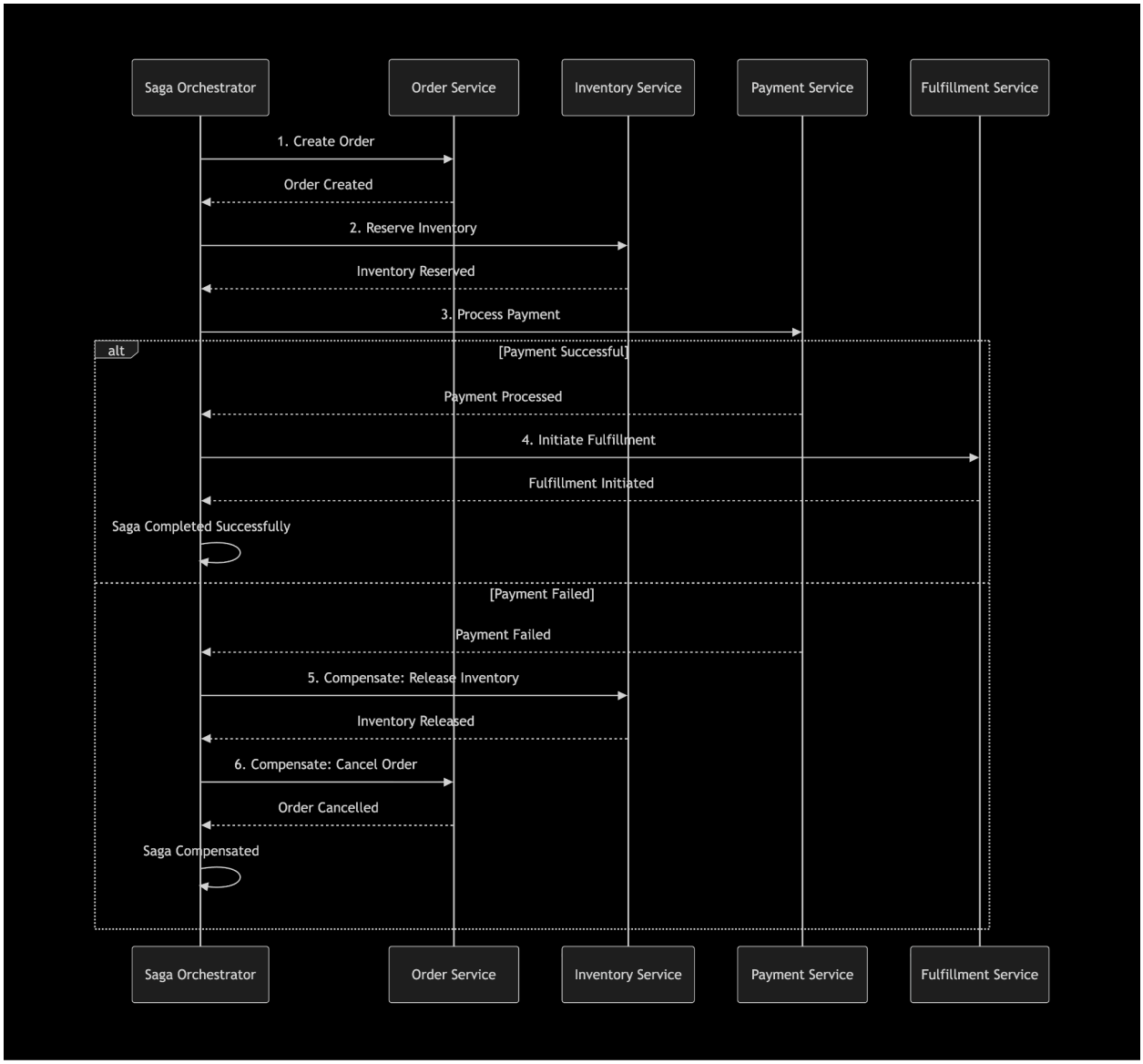
6.1 Synchronous Order Placement Flow



6.2 Asynchronous Order Processing Flow



6.3 Saga Pattern for Distributed Transactions


Saga Orchestrator
Order Service
Inventory Service
Payment Service
Fulfillment Service
1. Create Order
Order Created
2. Reserve Inventory
Inventory Reserved
3. Process Payment
alt
[Payment Successful]
Payment Processed
4. Initiate Fulfillment
Fulfillment Initiated
Saga Completed Successfully
[Payment Failed]
Payment Failed
5. Compensate: Release Inventory
Inventory Released
6. Compensate: Cancel Order
Order Cancelled
Saga Compensated
Saga Orchestrator
Order Service
Inventory Service
Payment Service
Fulfillment Service

7. DEPLOYMENT STRATEGY

7.1 Phase 1: Simple Deployment#

```
docker-compose.yml for local development
version: '3.8'
services:
  postgres:
    image: postgres:14
    environment:
      POSTGRES_DB: orderdb
      POSTGRES_USER: orderuser
      POSTGRES_PASSWORD: orderpass
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"
  order-service:
    build: .
    ports:
      - "8080:8080"
    environment:
      SPRING_PROFILES_ACTIVE: dev
      DB_URL: jdbc:postgresql://postgres:5432/orderdb
      REDIS_HOST: redis
    depends_on:
      - postgres
      - redis
volumes:
  postgres_data:
```

7.2 Phase 2: Production Deployment (AWS)#

AWS CloudFormation Template (Simplified)

```
Resources:
  OrderServiceECSCluster:
    Type: AWS::ECS::Cluster
    Properties:
      ClusterName: order-service-cluster

  OrderServiceTaskDefinition:
    Type: AWS::ECS::TaskDefinition
    Properties:
      Family: order-service
      Cpu: 1024
      Memory: 2048
      NetworkMode: awsvpc
      ContainerDefinitions:
        - Name: order-service
          Image: account.dkr.ecr.region.amazonaws.com/order-service:latest
          PortMappings:
            - ContainerPort: 8080
          Environment:
            - Name: SPRING_PROFILES_ACTIVE
              Value: prod
            - Name: DB_HOST
              Value: !GetAtt OrderDB.Endpoint.Address

  ApplicationLoadBalancer:
    Type: AWS::ElasticLoadBalancingV2::LoadBalancer
    Properties:
      Scheme: internet-facing
      Subnets: !Ref PublicSubnets

  # Auto Scaling Configuration
  OrderServiceAutoScaling:
    Type: AWS::ApplicationAutoScaling::ScalableTarget
    Properties:
      MinCapacity: 2
      MaxCapacity: 10
      ResourceId: !Sub service/${OrderServiceECSCluster}/${OrderServiceService}
      ScalableDimension: ecs:service:DesiredCount
      ServiceNamespace: ecs
```

7.3 Phase 3: Kubernetes Deployment

```
# order-service-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-service
  namespace: production
spec:
  replicas: 3
  selector:
    matchLabels:
      app: order-service
      tier: backend
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  template:
    metadata:
      labels:
        app: order-service
        tier: backend
    spec:
      containers:
      - name: order-service
        image: order-service:1.0.0
        ports:
        - containerPort: 8080
        env:
        - name: JAVA_OPTS
          value: "-Xmx1g -Xms512m -XX:+UseG1GC"
        resources:
          requests:
            memory: "512Mi"
            cpu: "250m"
          limits:
            memory: "1Gi"
            cpu: "500m"
        readinessProbe:
          httpGet:
            path: /actuator/health/readiness
            port: 8080
          initialDelaySeconds: 30
          periodSeconds: 10
        livenessProbe:
          httpGet:
            path: /actuator/health/liveness
            port: 8080
```

```
        initialDelaySeconds: 60
        periodSeconds: 10
---




# Service definition
apiVersion: v1
kind: Service
metadata:
  name: order-service
spec:
  selector:
    app: order-service
  ports:
  - port: 80
    targetPort: 8080
  type: ClusterIP
```

8. CAPACITY PLANNING

8.1 Traffic Estimates

Phase	Daily Orders	Peak Orders/Hour	Concurrent Users	Data Growth/Month
Launch	**1,000**	**100**	**500**	**500 MB**
Growth	**10,000**	**2,000**	**5,000**	**5 GB**
Scale	**100,000**	**20,000**	**50,000**	**50 GB**
Enterprise	**1,000,000**	**200,000**	**500,000**	**500 GB**

8.2 Infrastructure Sizing

Database Sizing

Phase 1 (Launch):
  Instance: db.t3.medium (2 vCPU, 4 GB RAM)
  Storage: 100 GB GP2
  Estimated Cost: $60/month
Phase 2 (Growth):
  Instance: db.r5.large (2 vCPU, 16 GB RAM)
  Storage: 500 GB GP2 + 2 Read Replicas
  Estimated Cost: $500/month
Phase 3 (Scale):
  Instance: db.r5.4xlarge (16 vCPU, 128 GB RAM)
  Storage: 2 TB GP3 + 4 Read Replicas
  Estimated Cost: $3,000/month
Phase 4 (Enterprise):
  Aurora PostgreSQL Global Database
  3 Regions, 16 vCPU per instance
  Estimated Cost: $15,000/month

8.2 Infrastructure Sizing

Database Sizing

Phase 1 (Launch):
  Instance: db.t3.medium (2 vCPU, 4 GB RAM)
  Storage: 100 GB GP2
  Estimated Cost: $60/month
Phase 2 (Growth):
  Instance: db.r5.large (2 vCPU, 16 GB RAM)
  Storage: 500 GB GP2 + 2 Read Replicas
  Estimated Cost: $500/month
Phase 3 (Scale):
  Instance: db.r5.4xlarge (16 vCPU, 128 GB RAM)
  Storage: 2 TB GP3 + 4 Read Replicas
  Estimated Cost: $3,000/month
Phase 4 (Enterprise):
  Aurora PostgreSQL Global Database
  3 Regions, 16 vCPU per instance
  Estimated Cost: $15,000/month

Application Server Sizing

Phase 1:
- Instances: 2 x t3.medium (2 vCPU, 4 GB RAM)
- Load: 30% CPU, 50% memory at peak
- Estimated Cost: $60/month

Phase 2:
- Instances: 4-8 x t3.large (2 vCPU, 8 GB RAM)
- Auto-scaling: 2-8 instances
- Estimated Cost: $200-800/month

Phase 3:
- Instances: 10-50 x c5.xlarge (4 vCPU, 8 GB RAM)
- Auto-scaling: 10-50 instances
- Estimated Cost: $1,500-7,500/month

8.3 Cost Projections

Component	Phase 1	Phase 2	Phase 3
Compute	$120/month	$1,000/month	$10,000/month
Database	$60/month	$500/month	$3,000/month
Cache	$0 (in-memory)	$100/month	$500/month
Message Queue	$0 (in-memory)	$50/month	$500/month
Monitoring	$0 (basic)	$200/month	$1,000/month
CDN/Network	$0	$100/month	$1,000/month
Total	$180/month	$1,950/month	$16,000/month

9. SECURITY CONSIDERATIONS

9.1 Security Layers

9.2 Security Implementation

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .cors().and()
            .csrf().disable() // For API, use token-based
            .sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            .and()
            .authorizeRequests()
                .antMatchers("/api/v1/public/**").permitAll()
                .antMatchers("/api/v1/orders/**").hasAuthority("ROLE_CUSTOMER")
                .antMatchers("/api/v1/admin/**").hasAuthority("ROLE_ADMIN")
                .anyRequest().authenticated()
            .and()
            .oauth2ResourceServer()
                .jwt()
                .jwtAuthenticationConverter(jwtAuthenticationConverter());

        // Add security headers
        http.headers()
            .contentSecurityPolicy("default-src 'self'")
            .and()
            .httpStrictTransportSecurity()
                .includeSubDomains(true)
                .maxAgeInSeconds(31536000);
    }

    // Multi-tenant security context
    @Bean
    public TenantFilter tenantFilter() {
        return new TenantFilter();
    }

    // Rate limiting
    @Bean
    public FilterRegistrationBean<RateLimitFilter> rateLimitFilter() {
        FilterRegistrationBean<RateLimitFilter> registration =
            new FilterRegistrationBean<>();
        registration.setFilter(new RateLimitFilter());
        registration.addUrlPatterns("/api/v1/*");
        registration.setOrder(Ordered.HIGHEST_PRECEDENCE);
        return registration;
```

```
        }
    }
```

9.3 PCI DSS Compliance Checklist

- Use tokenization for payment data
- Never store credit card numbers
- Encrypt sensitive data at rest (AES-256)
- Use TLS 1.2+ for all transmissions
- Implement access controls and audit trails
- Regular vulnerability scanning
- Penetration testing annually

10. MONITORING & OBSERVABILITY

10.1 Metrics Dashboard

Category	Key Metrics	Alert Threshold
Business	Orders/minute, Revenue/hour, Conversion rate	-20% from baseline
Application	P95 latency, Error rate, CPU usage	>200ms, >1% errors, >80% CPU
Database	Query latency, Connections, Cache hit ratio	>100ms, >80% connections, <90% hit ratio
Infrastructure	Memory usage, Disk I/O, Network throughput	>80% memory, >1000 IOPS

10.2 Spring Boot Actuator Configuration

application-monitoring.yml

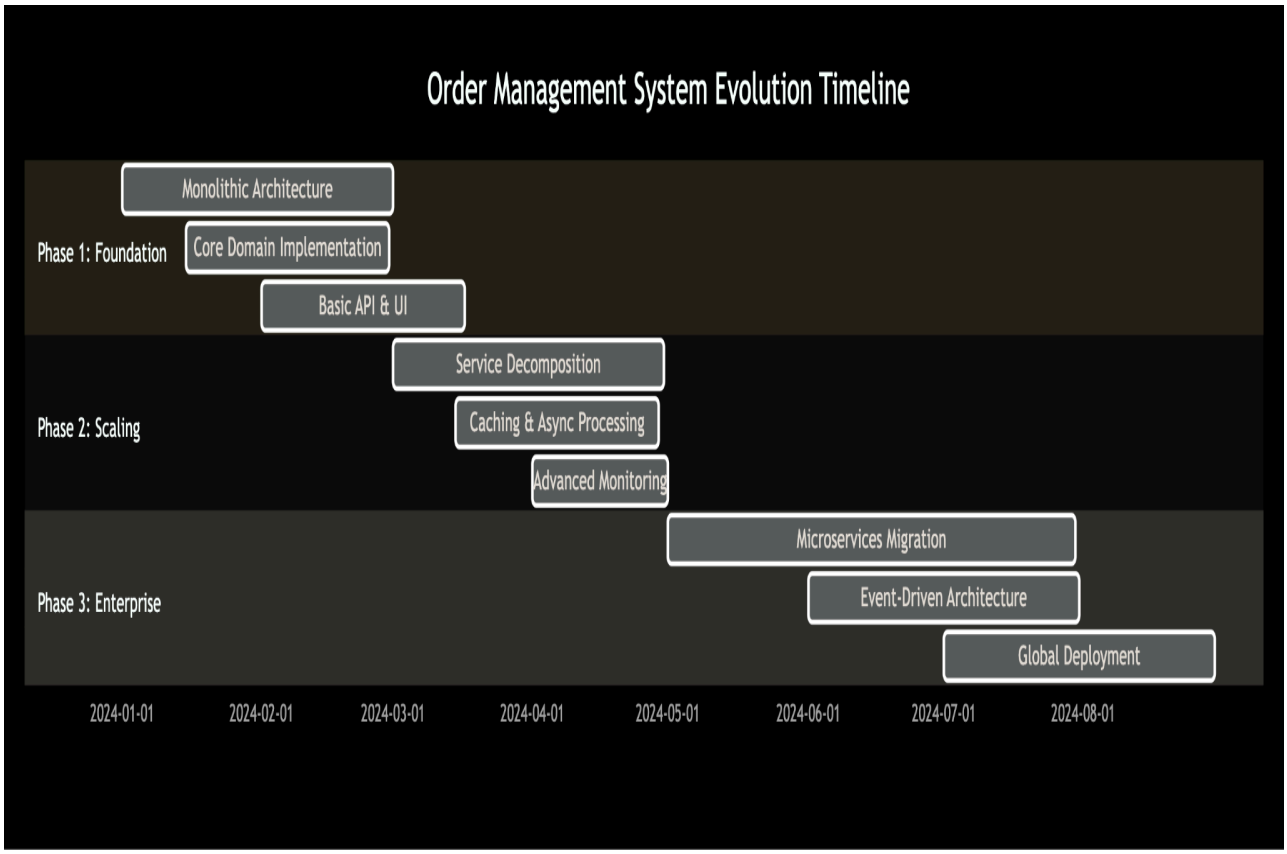
```yaml
management:
  endpoints:
    web:
      exposure:
        include: health,metrics,prometheus,info,loggers
      base-path: /actuator
  endpoint:
    health:
      show-details: always
      probes:
        enabled: true
  metrics:
    export:
      prometheus:
        enabled: true
    distribution:
      percentiles-histogram:
        http.server.requests: true
    tags:
      application: ${spring.application.name}
      environment: ${spring.profiles.active}
  tracing:
    sampling:
      probability: 1.0
```

10.3 Custom Health Indicators

```
@Component
public class OrderServiceHealthIndicator implements HealthIndicator {

    private final OrderRepository orderRepository;
    private final RedisTemplate<String, String> redisTemplate;

    @Override
    public Health health() {
        Health.Builder builder = Health.up();

        // Check database connectivity
        try {
            long orderCount = orderRepository.count();
            builder.withDetail("database.orders.count", orderCount);
        } catch (Exception e) {
            return Health.down()
                .withDetail("database.error", e.getMessage())
                .build();
        }

        // Check Redis connectivity
        try {
            String pong = redisTemplate.getConnectionFactory()
                .getConnection().ping();
            builder.withDetail("redis.status", "connected");
        } catch (Exception e) {
            return Health.down()
                .withDetail("redis.error", e.getMessage())
                .build();
        }

        return builder.build();
    }
}
```

11. EVOLUTION PHASES

11.1 Phase Timeline



11.2 Migration Checklist

Phase 1 → Phase 2 Migration Steps:

1. Implement API Gateway
2. Extract Inventory Service from monolith
3. Extract Payment Service from monolith
4. Implement service discovery
5. Add distributed caching
6. Implement message queue for async operations
7. Update CI/CD for multi-service deployment

Phase 2 → Phase 3 Migration Steps:

1. Containerize all services
2. Implement Kubernetes orchestration
3. Add service mesh (Istio/Linkerd)
4. Migrate to event streaming (Kafka)
5. Implement CQRS for read-heavy operations
6. Add multi-region database replication
7. Implement advanced observability

12. TRADE-OFFS & EXCLUSIONS

12. TRADE-OFFS & EXCLUSIONS

12.1 Deliberate Trade-offs

Decision	Chosen Approach	Alternatives Considered	Rationale
Initial Architecture	Modular Monolith	Microservices from start	Faster time-to-market, less operational overhead
Database	PostgreSQL	MongoDB, Cassandra	ACID compliance, relational data model fits orders
Caching Strategy	Cache-Aside	Read-Through, Write-Through	Simpler implementation, better control
Communication	REST + Async Events	gRPC, GraphQL	Wider tooling support, easier debugging
Deployment	EC2 → ECS → EKS	Serverless (Lambda)	More control, predictable performance

12.2 Exclusions & Future Considerations

Excluded from MVP:

- Real-time inventory updates across warehouses
- Advanced fraud detection
- AI/ML recommendations
- Mobile app push notifications
- Social media integration
- Voice commerce support
- AR/VR product visualization

Planned for Future:

- GraphQL API for flexible queries
- WebSocket for real-time order updates
- Blockchain for supply chain transparency
- Edge computing for low-latency regions
- Quantum-resistant encryption

13. RISK MITIGATION

13.1 Risk Assessment Matrix

Risk	Probability	Impact	Mitigation Strategy
Database downtime	Medium	High	Multi-AZ deployment, regular backups, read replicas
Payment gateway failure	Low	High	Multiple payment providers, manual override capability
Inventory inconsistency	Medium	High	Two-phase commit, regular reconciliation jobs
Security breach	Low	Critical	Regular audits, penetration testing, least privilege access
Scalability bottlenecks	High	Medium	Performance testing, auto-scaling, capacity planning

13.2 Disaster Recovery Plan

Recovery Objectives:

RTO (Recovery Time Objective): 1 hour

RPO (Recovery Point Objective): 5 minutes

Backup Strategy:

- Database: Automated daily full backup + continuous WAL archiving

- Application: Blue-green deployment with rollback capability

- Configuration: Version-controlled in Git

Recovery Procedures:

1. Identify failure point (monitoring alerts)

2. Failover to secondary region (if primary region down)

3. Restore from latest backup (if data corruption)

4. Validate data integrity

5. Resume operations with degraded mode if needed

Testing Schedule:

- Backup restoration test: Monthly

- DR drill: Quarterly

- Full failover test: Bi-annually

14. APPENDICES

14.1 Glossary

Term	Definition
OMS	Order Management System
SLA	Service Level Agreement
SLO	Service Level Objective
P95	95th percentile latency
RTO	Recovery Time Objective
RPO	Recovery Point Objective
DDD	Domain-Driven Design
CQRS	Command Query Responsibility Segregation

14.2 References

1. Spring Boot Documentation: https://spring.io/projects/spring-boot
2. PostgreSQL Documentation: https://www.postgresql.org/docs/
3. AWS Well-Architected Framework: https://aws.amazon.com/architecture/well-architected/
4. Domain-Driven Design by Eric Evans
5. Clean Architecture by Robert C. Martin

14.3 Revision History

Version	Date	Author	Changes
1.0	2024-12-25	Initial Team	Initial HLD Document
0.9	2024-12-20	Review Team	Architecture review updates
0.8	2024-12-15	Tech Lead	Capacity planning additions
0.7	2024-12-10	Architect	Security section completion

DOCUMENT APPROVAL

Role	Name	Signature	Date
Chief Architect			
Product Manager			
Engineering Manager			
Security Officer			
DevOps Lead			

Document Classification: Internal Use Only
Confidentiality Level: High
Retention Period: 7 years

This document represents the architectural decisions for the Order Management System and should be referenced for all technical decisions and implementation work.